KAPITEL 2

Erstes neuronales Netz mit PyTorch

In diesem Kapitel lernen Sie PyTorch weiter kennen. Wir erstellen ein richtiges neuronales Netz, das eine einfache, aber schon vertraute Aufgabe löst.

Das MNIST-Bilddatensatz

In meinem Buch *Neuronale Netze selbst programmieren* haben wir ein neuronales Netz entwickelt, das gelernt hat, Bilder von handgeschriebenen Ziffern zu klassifizieren.

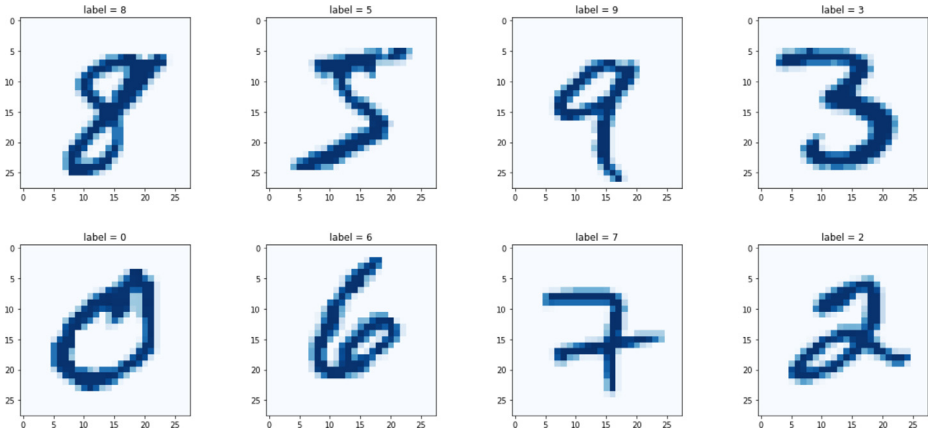


Abbildung 2-1: Beispiele für handgeschriebene Ziffern aus dem MNIST-Datensatz

Der *MNIST*-Datensatz ist ein bekannter Satz von Bildern, der oftmals herangezogen wird, um die Performance von Algorithmen für maschinelles Lernen zu messen und zu vergleichen. Er enthält 60.000 Bilder, die für das Training eines Modells für maschinelles Lernen gedacht sind, und 10.000 Bilder, um dessen Performance zu testen.

Die Bilder haben eine Größe von 28×28 Pixeln und sind monochrom, also nicht farbig. Die numerischen Werte der einzelnen Pixel geben an, wie hell oder dunkel

das jeweilige Pixel ist, und je nachdem, woher Sie die Daten erhalten, liegen die Werte zwischen 0 und 255.

Die MNIST-Daten abrufen

Gehen Sie in *Drive* zu Ihrem *Colab Notebooks*-Ordner, in dem Ihre Python-Notebooks gespeichert sind. Klicken Sie auf *Neu*, um einen neuen Ordner namens *mnist_data* zu erstellen.

Dieser *mnist_data*-Ordner sollte sich im selben Ordner wie Ihre Notebooks befinden (siehe Abbildung 2-2).

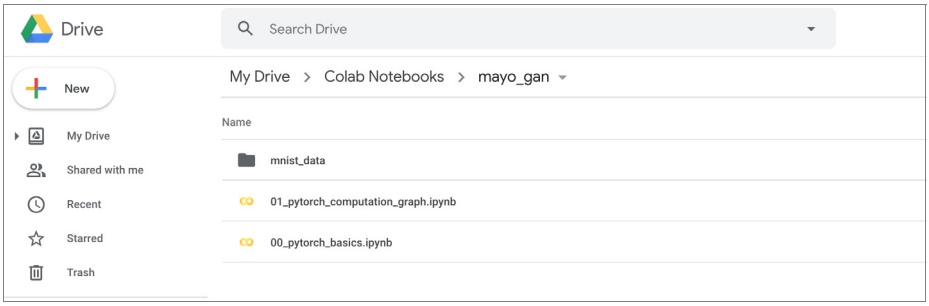


Abbildung 2-2: Der neu erstellte Ordner »mnist_data«

Laden Sie die MNIST-Daten über die folgenden Links auf Ihren Computer herunter:

- Trainingsdaten: *https://pjreddie.com/media/files/mnist_train.csv*
- Testdaten: *https://pjreddie.com/media/files/mnist_test.csv*

Nachdem Sie diese beiden Dateien heruntergeladen haben, laden Sie sie in den Ordner *mnist_data* hoch. Gehen Sie hierfür zum Ordner *mnist_data*, klicken Sie auf die Schaltfläche *Neu* und wählen Sie den Befehl *Dateien hochladen*.

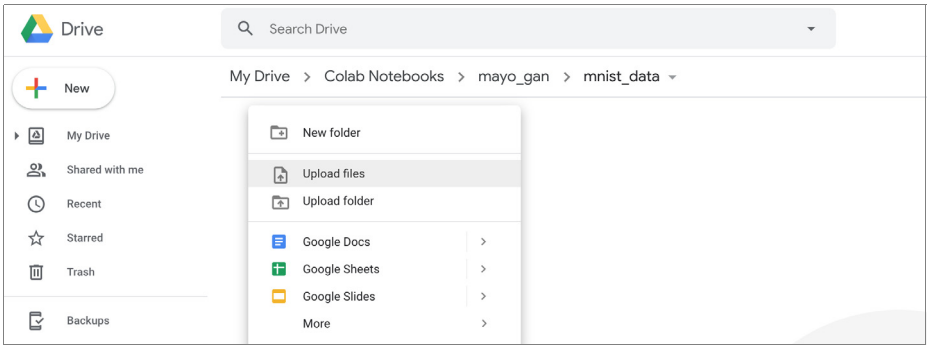


Abbildung 2-3: Die heruntergeladenen MNIST-Daten in den Ordner »mnist_data« hochladen

Nach kurzer Zeit sollten die beiden Dateien im Ordner *mnist_data* aufgelistet werden.

Ein Blick auf die Daten

Es ist eine gute Angewohnheit, alle neuen Daten vorab zu inspizieren, um ein Gefühl dafür zu bekommen, bevor Sie sie mit Tools und Algorithmen attackieren!

Wir müssen unsere hochgeladenen Daten für unseren Python-Code zugänglich machen. Hierfür mounten wir unser Laufwerk (Drive), sodass es als Ordner erscheint.

Beginnen Sie ein neues Notebook und führen Sie den folgenden Code aus:

```
# Laufwerk mounten, um auf Datendateien zuzugreifen
from google.colab import drive
drive.mount('./mount')
```

Sie werden aufgefordert, auf einen Link zu klicken, der Sie zu einem neuen Register bringt. Auf dieser Registerkarte sollen Sie das Konto bestätigen und Berechtigungen für das Laufwerk gewähren, das gemountet und zugänglich gemacht werden soll. Daraufhin erhalten Sie einen Code, den Sie kopieren und in die Notebook-Zelle einfügen (siehe Abbildung 2-4).

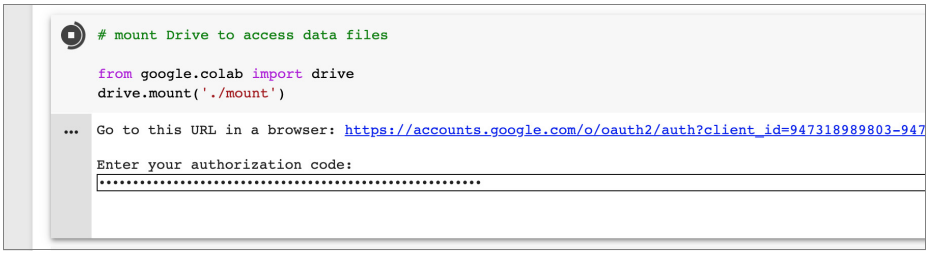


Abbildung 2-4: Den Autorisierungscode kopieren und einfügen, um das Laufwerk zu mounten und zugänglich zu machen

Sobald das erledigt ist, erhalten Sie eine Bestätigung, dass das Drive gemountet wurde und für Ihren Python-Code aus dem Ordner *./mount* sichtbar ist.

Unsere MNIST-Datendateien liegen im CSV-Format vor, d.h. Zeilen mit Werten, die jeweils durch Kommata getrennt sind (engl. *Comma Separated Values*). Es gibt viele Wege, um die Daten zu laden und anzuzeigen. Wir verwenden die *Pandas*-Bibliothek, da sie die Aufgabe wirklich einfach macht.

Importieren Sie in einer neuen Zelle die Pandas-Bibliothek:

```
# Pandas importieren, um CSV-Dateien zu lesen
import pandas
```

In der nächsten Zelle nutzen wir Pandas, um die Trainingsdaten in einen Dataframe zu laden. Der folgende Code ist eigentlich eine einzelne Zeile. Er ist nur so lang, weil der Dateipfad zur Datei *mnist_train.csv* so lang ist:

```
df = pandas.read_csv('mount/My Drive/Colab Notebooks/myo_gan/mnist_data/
mnist_train.csv', header=None)
```

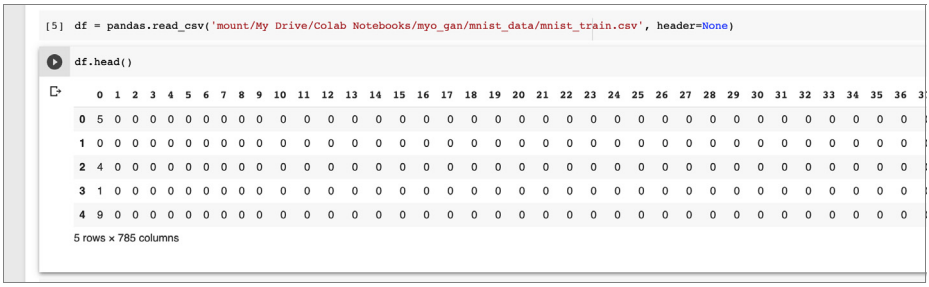
Kontrollieren Sie, ob der von Ihnen verwendete Dateipfad auch dem entspricht, in dem Sie Ihre Dateien abgelegt haben. Mein eigener Code und die Daten befinden sich in einem Ordner *myo_gan* innerhalb von *Colab Notebooks*.

Ein *Pandas-Dataframe* ist einem NumPy-Array ähnlich, besitzt aber zusätzliche Features wie zum Beispiel benannte Spalten und Zeilen sowie viele Komfortfunktionen wie etwa solche zum Summieren und Filtern von Daten.

So können wir mit der Funktion `head()` einen Blick auf die Spitze eines großen Dataframes werfen:

```
df.head()
```

Als Ergebnis werden die ersten fünf Zeilen des Dataframes angezeigt.



```
[5] df = pandas.read_csv('mount/My Drive/Colab Notebooks/myo_gan/mnist_data/mnist_train.csv', header=None)

df.head()
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	3
0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
2	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
3	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	
4	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	

5 rows × 785 columns

Abbildung 2-5: Die ersten fünf Zeilen eines großen Dataframes

Jede Zeile der MNIST-Daten besteht aus 785 Werten. Die erste Zahl ist die Ziffer, die das Bild darstellen soll, die restlichen 784 Zahlen sind die Pixelwerte für das 28×28-Bild. Mit der Funktion `info()` erhalten Sie einen Überblick über den Dataframe.

```
[9] df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, 0 to 784
dtypes: int64(785)
memory usage: 359.3 MB
```

Abbildung 2-6: Überblicksinformationen für einen Dataframe

Aus der Zusammenfassung erfahren wir, dass der Dataframe 60.000 Zeilen umfasst. Das sind die 60.000 Trainingsbilder. Außerdem wird bestätigt, dass eine Zeile aus 785 Werten besteht.

Wir visualisieren nun die Daten, indem wir aus einer Zeile mit Pixelwerten ein tatsächliches Bild erzeugen.

Hierzu greifen wir auf die vielseitige Bibliothek `matplotlib` zurück. Aktualisieren Sie die Zelle, in der die Bibliotheken importiert werden, um das Paket `pyplot` aus der Bibliothek `matplotlib` einzubinden:

```
# Pandas importieren, um CSV-Dateien zu lesen
import pandas
```

```
# matplotlib importieren, um Bilder anzuzeigen
import matplotlib.pyplot as plt
```

Führen Sie die aktualisierte Zelle erneut aus, um `pyplot` verfügbar zu machen. Sehen Sie sich den folgenden Code an:

```
# Daten von Dataframe abrufen
row = 0
data =  df.iloc[row]

# Das Label ist der erste Wert.
label = data[0]

# Die Bilddaten sind die restlichen 784 Werte.
img = data[1:].values.reshape(28,28)
plt.title("label  =  " +  str(label))
plt.imshow(img, interpolation='none',  cmap='Blues')
plt.show()
```

Der erste Teil des Codes wählt aus den MNIST-Daten das Bild aus, das uns interessiert. Das erste Bild, das in der ersten Zeile enthalten ist, wird mit `row = 0` ausgewählt. Die Codezeile `df.iloc[row]` holt die erste Zeile aus dem Datensatz und weist sie der Variablen `data` zu.

Im nächsten Codeabschnitt rufen wir die erste Zahl aus dieser Zeile ab und nennen sie `label`, da sie die jeweilige Ziffer kennzeichnet (von engl. label – Kennzeichen).

Der dritte Teil des Codes übernimmt die restlichen 784 Werte aus dieser Zeile, formt sie in ein quadratisches Array von 28 Zeilen und 28 Spalten um und weist sie einer Variablen `img` (von engl. image – Bild) zu. Das Array wird dann als Bitmap gezeichnet, wobei der Titel das Label zeigt, das wir aus den Daten extrahiert haben. Für die Funktion `imshow()`, die das Bitmap zeichnet, sind viele Optionen definiert. Wir verwenden hier die beiden Optionen, die die Farbpalette festlegen und `pyplot` anweisen, die Pixel nicht zu glätten.

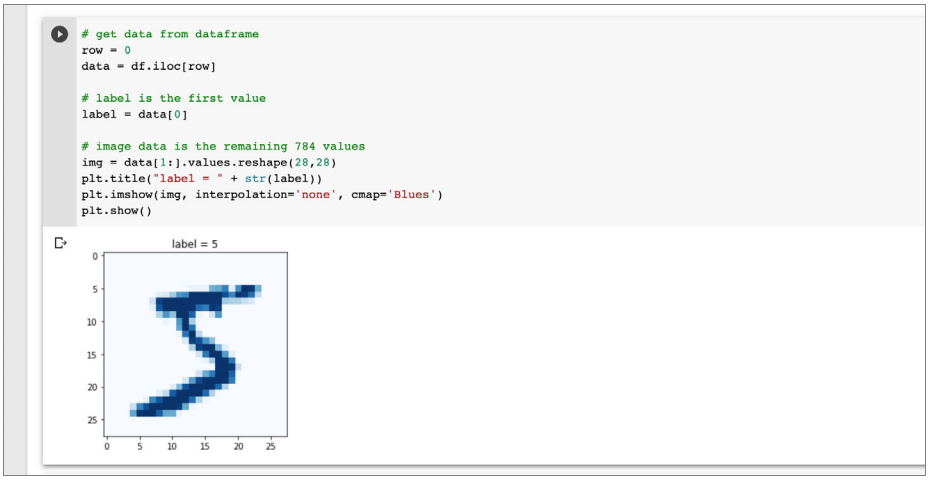


Abbildung 2-7: Darstellung der Bilddaten für die erste Ziffer

Abbildung 2-7 zeigt das Ergebnis mit dem ersten Bild im MNIST-Trainingsdatensatz. Es sieht wie eine Fünf aus, und das Label bestätigt, dass es eine Fünf sein soll.

Experimentieren Sie mit anderen Bildern aus dem Datensatz, indem Sie den Zeilenwert `row` ändern und die Zelle erneut ausführen. Wenn Sie zum Beispiel `row = 13` probieren, sollten Sie ein Bild sehen, das wie eine Sechs aussieht.

Der Code, den wir bisher entwickelt haben, ist online unter folgendem Link zu finden:

- *https://github.com/makeyourownneuralnetwork/gan/blob/master/02_mnist_data.ipynb*

Ein einfaches neuronales Netz

Bevor wir uns eingehend damit befassen, Code für ein neuronales Netz zu schreiben, treten wir einen Schritt zurück und machen uns ein Bild davon, was wir zu erreichen versuchen. Abbildung 2-8 zeigt unseren Ausgangspunkt und wo wir letztlich hinwollen.

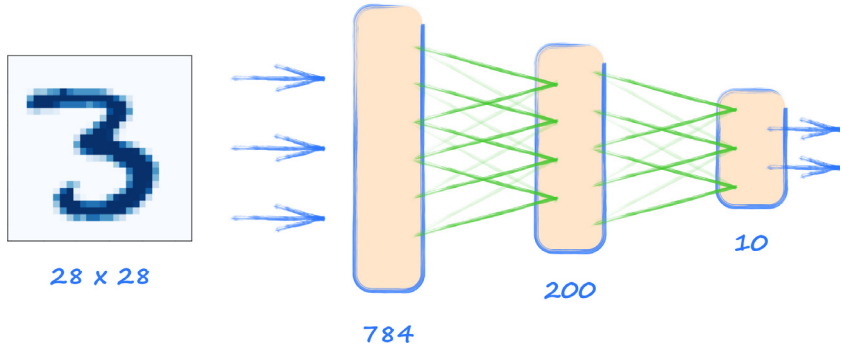


Abbildung 2-8: Ausgangspunkt und Ziel für ein neuronales Netz

Den Ausgangspunkt bildet ein MNIST-Bild, das 28 mal 28 oder 784 Pixelwerte umfasst. Das heißt, die erste Schicht unseres neuronalen Netzes muss aus 784 Knoten bestehen. Bei der Größe der Eingabeschicht haben wir kaum eine Wahl.

Die Größe der letzten Schicht, der Ausgabeschicht, können wir in gewissen Grenzen beeinflussen. Diese Schicht muss in der Lage sein, die Frage zu beantworten: »Welche Ziffer ist das?« Die Antwort könnte eine der Ziffern von 0 bis 9 sein, was 10 verschiedene Möglichkeiten bedeutet. Der einfachste Ansatz sieht einen Knoten für jede dieser zehn möglichen Klassen vor.

Mehr Entscheidungsfreiheit haben wir bei der mittleren – der versteckten – Schicht. Da wir vor allem etwas über PyTorch lernen und nicht die beste Größe finden wollen, verwenden wir das, was wir in *Neuronale Netze selbst programmieren* gelernt haben, und wählen für den Anfang eine Größe von 200.

Alle Knoten in einer Schicht sind mit jedem Knoten in der nächsten Schicht verbunden. Man spricht dann auch von *vollständig verbundenen Schichten*.

Diesem Bild fehlt ein wichtiges Element. Wir müssen eine Aktivierungsfunktion auswählen, die auf die Ausgaben von versteckter und Ausgabeschicht angewendet wird. In *Neuronale Netze selbst programmieren* haben wir die s-förmige *logistische Funktion* verwendet. Der Einfachheit halber bleiben wir dabei.

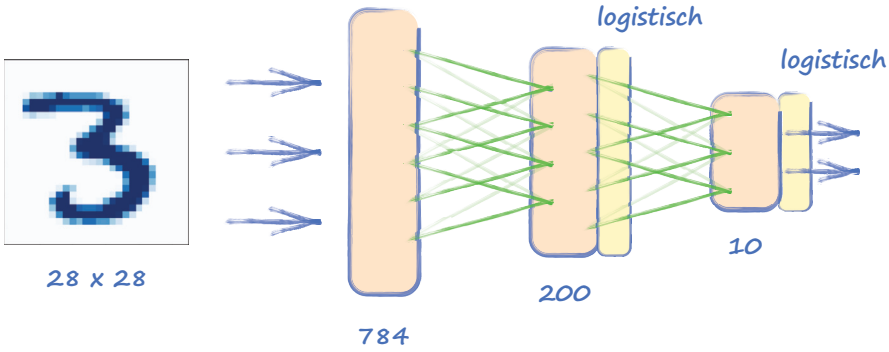


Abbildung 2-9: Das einfache neuronale Netz mit logistischen Funktionen in der versteckten und der Ausgabeschicht

Wir können nun diese *Architektur* des neuronalen Netzes in PyTorch-Code umsetzen. Mit PyTorch lassen sich neuronale Netze einfacher erstellen und ausführen, da diese Bibliothek eine Menge Arbeit hinter den Kulissen erledigt. Damit das funktioniert, müssen wir uns an das Codierungsmuster von PyTorch halten.

Wenn wir eine Klasse für ein neuronales Netz erstellen, müssen wir von der PyTorch-eigenen Klasse `torch.nn` erben. Das bringt eine Menge an PyTorch-Maschinerie mit sich, beispielsweise das automatische Erstellen des Berechnungsgraphen, die Behandlung der Gewichte und die Aktualisierung der Gewichte während des Trainings.

Legen Sie ein neues Notebook an und importieren Sie sowohl `torch` als auch `torch.nn`:

```
# Bibliotheken importieren
import torch
import torch.nn as nn
```

Das Modul `torch.nn` wird als `nn` importiert, was zur Namenskonvention geworden ist.

Erstellen wir nun die Klasse für unser neuronales Netz. Der folgende Code zeigt eine Klasse, die wir `Classifier` genannt haben und die von `nn.Module` erbt:

```
class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()
```

Die spezielle Funktion `__init__(self)` wird aufgerufen, wenn ein Objekt aus einer Klasse erstellt wird. Oftmals dient sie dazu, ein Objekt einzurichten und es einsatzbereit zu machen. Vielleicht haben Sie schon gehört, dass sie *Konstruktor* genannt wird, was ein recht aussagekräftiger Name ist. Hier geschieht diese Einrichtung mit `super().__init()` – das sieht etwas eigenartig aus, ruft aber einfach den Konstruktor der übergeordneten Klasse auf. Somit wird `nn.Module` von PyTorch die Klasse `Classifier` für uns einrichten. Toll, oder?

Wir definieren nun die Architektur des neuronalen Netzes. Hierfür gibt es verschiedene Herangehensweisen. Bei einfachen Netzen können wir mit `nn.Sequential()` eine Liste der Netzelemente bereitstellen. Die Elemente müssen in der Reihenfolge erscheinen, in der die Informationen sie durchlaufen sollen.

```
class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()

        # Schichten des neuronalen Netzes definieren
        self.model = nn.Sequential(
            nn.Linear(784, 200),
            nn.Sigmoid(),
            nn.Linear(200, 10),
            nn.Sigmoid()
        )
```

In der Funktion `nn.Sequential()` sind die folgenden Elemente definiert:

- `nn.Linear(784, 200)` ist eine vollständig verbundene Zuordnung von 784 Knoten auf 200 Knoten. Dieses Element enthält die Gewichte für die Verbindungen zwischen den Knoten, die während des Trainings aktualisiert werden.
- `nn.Sigmoid()` wendet die s-förmige logistische Aktivierungsfunktion auf die Ausgaben des vorherigen Elements an, in diesem Fall die 200 Knoten.
- `nn.Linear(200, 10)` bildet 200 Knoten auf 10 Knoten ab. Dieses Element enthält die Gewichte für die Verbindungen zwischen der mittleren, versteckten Schicht und der letzten Schicht mit den 10 Ausgabeknoten.
- `nn.Sigmoid()` wendet die s-förmige logistische Aktivierungsfunktion auf die Ausgabe der 10 Knoten an. Das Ergebnis ist die endgültige Ausgabe des Netzes.

Vielleicht fragen Sie sich, warum die Funktion `nn.Linear()` so genannt wird. Das liegt daran, dass sie eine lineare Funktion der Form `Ax + B` auf die Eingabewerte anwendet und an die Ausgabe liefert. Das `A` bezeichnet die Verbindungsgewichte, und das `B` kann man als Schwellenwert (engl. Bias) betrachten. Beide Parameter werden während des Trainings aktualisiert. Man spricht auch von *lernbaren Parametern*.

Für das neuronale Netz haben wir die Elemente definiert, die die Informationen in Vorwärtsrichtung weitergeben, doch wir haben noch nicht definiert, wie das Netz den Fehler ermittelt und damit dann die lernbaren Parameter des Netzes aktualisiert.

Es gibt viele Möglichkeiten, den Fehler für das Netz zu definieren. Für gängige Optionen stellt PyTorch komfortable Funktionen bereit. Eine der einfachsten berechnet den *mittleren quadratischen Fehler*. Dabei werden die Differenzen zwischen den tatsächlichen und den gewünschten Ausgaben an jedem der Ausgabeknoten quadriert, und daraus wird dann der Mittelwert gebildet. PyTorch realisiert dies mit `torch.nn.MSELoss()`.

Im Konstruktor können wir diese Fehlerfunktion auswählen und ihr einen Namen geben.

```
# Verlustfunktion erzeugen
self.loss_function = nn.MSELoss()
```

Gelegentlich werden die Begriffe *Fehlerfunktion* und *Verlustfunktion* gleichberechtigt nebeneinander verwendet, und oftmals ist das auch in Ordnung. Um genau zu sein: Der *Fehler* ist die einfache Differenz zwischen einer gewünschten und der tatsächlichen Ausgabe, und der *Verlust* wird aus dem Fehler in einer Weise berechnet, die für das zu lösende Problem zweckmäßig ist.

Wir müssen diesen *Fehler* – oder richtiger gesagt, den *Verlust* – verwenden, um die Verbindungsgewichte des Netzes zu aktualisieren. Auch hierfür gibt es mehrere Verfahren, und PyTorch bietet Funktionen für gängige Optionen. Beginnen wir mit der einfachen Version, die in *Neuronale Netze selbst programmieren* entwickelt wurde: dem sogenannten *stochastischen Gradientenabstieg* (SGD, von engl. *Stochastic Gradient Descent*) mit einer Lernrate von `0.01`.

```
# Optimierer erstellen, dafür einfachen stochastischen Gradientenabstieg verwenden
self.optimiser = torch.optim.SGD(self.parameters(), lr=0.01)
```

Interessant ist, dass wir alle lernbaren Parameter an den SGD-Optimierer übergeben. Diese sind über die Methode `self.parameters()` zugänglich, die die Maschinerie von PyTorch für uns erzeugt.

Um Informationen durch das Netz zu leiten, geht PyTorch von einer Methode `forward()` aus. Wir müssen also eine solche Methode erzeugen, die allerdings minimal ausfallen darf:

```
def forward(self, inputs):
    # Modell einfach ausführen
    return self.model(inputs)
```

Hier nehmen wir die gegebenen Eingaben und übergeben sie an die Methode `self.model()`, die wir oben mithilfe von `nn.Sequential()` definiert haben. Die Ausgabe des Modells wird einfach an den Aufrufer der Methode `forward()` zurückgegeben.

Gönnen wir uns eine Pause und gehen wir noch einmal durch, was wir bisher erledigt haben:

- Wir haben eine Klasse für ein neuronales Netz erstellt, die von `nn.Module` erbt. Durch das Vererben von `nn.Module` bekommen wir viel von der Maschinerie mit, die für das Trainieren eines neuronalen Netzes erforderlich ist.

- Wir haben die Elemente des neuronalen Netzes definiert, durch die Informationen fließen. Die Wahl ist auf die Methode `nn.Sequential` gefallen, da sie für einfache Netze kurz und bündig ist.
- Wir haben die Verlustfunktion und die Optimierermethode definiert, um die lernbaren Parameter des Netzes zu aktualisieren.
- Schließlich haben wir eine `forward()`-Funktion hinzugefügt, die PyTorch erwartet, um Informationen durch das Netz zu leiten.

Unsere Klasse für das neuronale Netz sieht jetzt folgendermaßen aus:

```
# Klassifiziererklasse

class Classifier(nn.Module):

    def __init__(self):
        # Übergeordnete PyTorch-Klasse initialisieren
        super().__init__()

        # Schichten des neuronalen Netzes definieren
        self.model = nn.Sequential(
            nn.Linear(784, 200),
            nn.Sigmoid(),
            nn.Linear(200, 10),
            nn.Sigmoid()
        )

        # Verlustfunktion erstellen
        self.loss_function = nn.MSELoss()

        # Optimierer mit einfachem stochastischem Gradientenabstieg erstellen
        self.optimiser = torch.optim.SGD(self.parameters(), lr=0.01)

        pass

    def forward(self, inputs):
      # Modell einfach ausführen
      return self.model(inputs)
```

Wie trainieren wir das Netz? Brauchen wir eine Funktion `train()`, so wie wir eine Funktion `forward()` haben? Eigentlich verlangt PyTorch keine `train()`-Methode. Es bleibt uns überlassen, unseren Code für das Training zu strukturieren.

Wir wollen unseren eigenen Code einfach und konsistent halten und erstellen neben der Methode `forward()` eine Methode `train()`, die die Methode `forward()` begleitet.

Eine `train()`-Methode benötigt sowohl die *Eingänge* in das Netz als auch die gewünschten *Ausgänge* des *Ziels*, damit sie diese mit dem tatsächlichen Ausgang vergleichen und den *Verlust* berechnen kann.

```
def train(self, inputs, targets):
    # Den Ausgang des Netzes berechnen
    outputs  = self.forward(inputs)
```